Haile Quals day 3

What do we already know about the cognitive basis(es) of learning computer programming languages? Based on your other readings on learning and memory more generally, what might be missing from our current knowledge?

Introduction

Learning a computer programming language is a difficult task that could be examined as a combination of acquiring domain knowledge, learning how to apply learned information and problem-solving skills (Lye and Koh, 2014; Federenko et al., 2019; Robins et al., 2003; Mayer, 1981). Additionally, thinking about programming language learning as being hierarchically organized where different aspects are learned and simpler skills are combined to be used to solve more complex problems, might allow us to understand how cognitive functions may be applied (Robins et al., 2003; Federenko et al. 2019). Like all cognitive skills, computer programming is supported by fundamental declarative and procedural learning and memory systems and there are clearly defined skill acquisition models that are not discussed by programming learning researchers. On the other hand, addressing the problem of cognitive load during learning is missing from cognitive researchers even while many learning researchers broadly discuss how to ease cognitive load during learning. Lastly, while some research examines programming skill learning from the point of view of learning and memory systems, we do not yet have a good understanding of where individual differences might arise. This makes improving teaching programming languages more challenging.

Learning to program requires acquiring domain knowledge

Understanding computer programming learning will benefit from first dividing the set of skills needed into 1) acquiring domain knowledge, 2) domain-general problem-solving, and 3) domain-specific problem-solving skills. Lye and Koh (2014) discuss a skill called computational thinking that is acquired during programming learning. They describe computational thinking as consisting of learning programming concepts, problem-solving skills and new perspectives that relate the learners to technology and other programmers. They forward computational learning as a skill worth teaching to students so that students may apply the skills in their daily lives. But we will use the computational thinking concept to examine programming language learning and focus on the first two dimensions. Before we continue, it should be clarified that research on programming pedagogy, perhaps usefully, dissociates programming skill as summarized above but research on cognition shows that they are not so cleanly dissociable. This means that the domain-specific declarative knowledge and the problem-solving know-how develop together, and a transformation of knowledge representation occurs as skill advances to expert level (Anderson, 1982).

Computer programming languages come with a vast set of concepts most of which are unique to programming languages. Learners must leverage the use of strategies that improve retention of these new concepts. Some mnemonic strategies for better acquisition of declarative memory are discussed below.

For-loops, functions, operators, if-then statements, variables, etc., are concepts that represent the building blocks of programs. Declarative memory is a vast storage of chunks of interconnected facts. Declarative memory seems to strive to represent information that is likely to be relevant to the learner and is driven by how often the learner is exposed to it in their environment. This makes logical sense as information that is predicted to be useful should receive an increased level of activation, enabling better recall or availability if it is needed (Anderson, 2000). Learning new programming concepts in a durable way, therefore, is better accomplished if the learner reviews the new programming concepts periodically.

Most of these concepts will be foreign to novice programmers but, fortunately, it is safe to assume that most learners will have mastered how to use basic mathematical operators like addition, multiplication and subtraction which could provide a schema to add new information to. In other words, there already exists a set of general, transferable knowledge that can be built upon (Mayer 1981). New concepts are better learned when there are linking points that already exist in the learners memory (Anderson, 2000), mathematical concepts are just one example of many. It may be helpful to think about declarative memory as a web of interconnected concepts with open attachment ends that are ready to connect to new information.

Learning to program by immediately applying new concepts to solve problems is very advantageous because it allows for deeper processing. For instance, reading off and rehearsing a list of useful Python functions several times is not as helpful as writing simple code that uses the functions. Practicing with writing code will allow for elaborative processing which amplifies acquisition of the concept through formation of additional cues for retrieval, according to a well-researched depth-of-processing theory (Anderson, 2000).

learning to program requires domain-general and domain-specific Problem-solving skill

Representing new and useful programming concepts in declarative memory is useful but it is merely a very early stage in programming skill development. There is a very murky process that most learning and teaching researchers call *understanding*. Understanding is the ability to manipulate and apply the acquired information towards new problem-solving situations (Mayer, 1981). Great folly though this might be, I will attempt to distinguish understanding from domain-specific programming problem-solving skills. We can call it a more domain-general problem-solving skill set. Like Mayer (1981), Anderson (1982) suggests that skill learning involves interpreting recently acquired declarative information about how to perform a task into efficient and automatic processes that we can observe in expert skill performance. In Anderson's model of cognitive skill learning, all skills start out as declarative information. Through practice, these declarative representations are *interpreted* into what he calls procedural rules. Procedural rules

dictate a set of rules for responding to specific environmental cues or situations. In the early parts of learning a skill we may produce many individual procedures for responding to specific situations. To take a programming example, imagine a novice learner who wants to print the message "Thrive Through" in their programming environment. They might have different procedures for opening the Python development environment that contain actions for double clicking an icon, creating a new file by selecting that available functionality, and more procedures for typing in the correct syntax to print the message. All these procedures would depend on the recently acquired declarative information that will guide this behavior. Anderson describes a higher level of developed skill where these individual procedures are consolidated into more efficient procedures that would not depend on directives stored in declarative memory. The new procedure, for example, would simply contain the instructions "create a new file in python environment and type "print("Thrive Through")"". This is possible because sufficient practice will encode efficient patterns of motor actions that accomplish the end goal. However, this is a representation of expert skill performance and different learners employ different strategies to apply their acquired information to problem solving situations (Anderson, 1982).

In an experiment that tested the role of examples in learning, Anderson et al. (1999) described how subjects formed intuitions from the explicit examples on how to solve specific problems. These learners used a mixture of strategies that vary in levels of efficiency. For clarity, a less efficient strategy is slow and requires laborious, multi-step solving of a problem. Four specific strategies were discussed. In this experiment subjects were tasked with learning statements like "Skydiving was practiced on Saturday at 5 pm and Monday at 4 pm" that contained specific rules. In the above example, the statement contains a rule for how often and at what time of day the specific sport is practiced. Subjects were then given one day and time for a specific sport and were asked to predict the second day and time practice would occur. Learners, rapidly but limitedly, directly recalled responses to problems they have seen before. They also analogically extrapolated a solution by recalling a similar example, which is taxing and inefficient. After some practice, they were able to extract a declarative rule and applied it without referring to examples in memory. And lastly, with more practice, they were able to form very efficient procedural rules akin to the procedural rules exemplified above. This is the clearest characterization of different strategies that could be utilized by new learners to apply learned declarative information to new problems. The study had observed that a mixture of these strategies was used by learners throughout the experimental sessions. This shows that different avenues maybe taken to arrive at expert performance or understanding. To my knowledge, there has been little research done on how learners develop and apply strategies to solve programming problems.

Moving on to domain-specific problem-solving skills that are acquired, it will be helpful to think of a programming task as having a hierarchical structure. There are a few reasons why this will be helpful. Firstly, Novice and expert programmers solve different types of problems (Robins et al., 2003). Novice learners are concerned with composing statements of code that work as intended and a lot of the issues arise simply from incorrect syntax, wrong order of

operations, an inability to translate concepts into code or not knowing what functions or operations would be best suited to the task at hand. Experts, on the other hand, have mastered the above lower-tier skills needed to make a piece of code work. Most of their concerns now lie with solving more abstracted problems that may require, for example, integrating multiple scripts of code. To take an analogical example, if novice writers are still learning to spell words and construct sentences in English, the experts would be tackling how to use multiple pages of text to argue a specific philosophical position and consider multiple ways constructing arguments.

There are far too many programming-specific skills that are built from simpler components to enumerate (like debugging code, building recursive code, formatting data, object-oriented programming, etc.), and different sets appear at different levels of this problem-solving hierarchy, but there is some relief in the fact these skills are cognitively represented by the same learning and memory mechanisms at all levels of the hierarchy. This idea is supportable by Anderson's model of skill progression described above, except now the declarative stores would likely be used for learning the larger moving parts of the problem instead of simple programming syntax. Lee and Anderson (2001) have also shown that efficiencies in skill demonstration developed at higher levels of an air traffic controller task like, triaging and landing incoming simulated airplanes, resembled, and were developed from, learning at more basic levels of the task like pressing the correct keys and attending to relevant portions of the screen. Studies that track the development of proficiencies at lower levels to predict how higher-level proficiencies are developed are largely lacking in the programming language learning literature.

The murky problem of cognitive load

A lot of the research on how to teach programming languages also discuss the need to manage cognitive load (Lye and Koh, 2014; Robins et al., 2003). The idea of cognitive load is not clearly defined but it hints at specific limitations in information processing that we all have. A well-documented, albeit still murky, source of such a limitation, that has implications on numerous cognitive tasks is working memory capacity. We quickly realize our limits in maintaining information when we attempt to hold, for instance, a large sequence of numbers in memory, like a credit card number, to make a purchase online. But it is not clear yet how this capacity limitation arises. Depending on who you ask, this capacity limitation maybe nothing more than poor mnemonic strategy — one could hold more figures in mind if they used a chunking strategy. Cowan (1999) argues that there may not be physiologically realized limitations. Some theories suggest that capacity limitations arise due to poor filtration of irrelevant features via attention and the items held in mind might interfere with each other resulting in loss of some of the items (Cowan, 1999).

Engaging in learning a new skill requires deployment of attention and manipulation of relevant items in working memory. Additionally, learning a new skill involves managing capacity limited cognitive resources, just as much as learning new concepts. Most studies of skill acquisition describe an ultimate stage of skill that is automatic and effortless (e.g., Anderson,

1982; Schneider and Chein, 2003; Tenison and Anderson, 2015). This effortless-ness of skilled performance arises because conscious direction of attentional resources is no longer required. Lee and Anderson (2001) have shown that more experienced subjects were more efficient and had reduced attending to irrelevant stimuli. There are additional theories that show that attention deployment is "proceduralized" and is automatically deployed to where the most relevant features of the stimuli are (Ramamurthy and Blaser, 2017; Haile et al., 2016).

However, before learners develop efficient use of their attentional and working memory resources new information must be presented to them in small portions and incrementally in a way that does not overwhelm them (Lye and Koh, 2014; Robins et al., 2003). An example of Such strategies that aid in learning and manage to reduce cognitive load involve interpreting abstract concepts into concrete examples that learners already know.

A final point on cognitive load concerns natural languages. Federenko et al., (2019) and Prat et al. (2020) argue that programming languages share similar structures to natural languages when decomposed. Language is one of the most complex and well-developed skills we possess, therefore the more similar a programming language is to a natural language (usually just the English language) the more sparing it is of cognitive capacity. Lye and Koh (2014), for example, describe that older programming languages represented information more like computers than how humans represented information, and were therefore difficult to learn.

Individual differences in learning are poorly addressed

It will not escape the avid reader that individual differences are not at all discussed in the preceding pages. Individual differences are very rarely, if at all, addressed in the programming language learning research, or even the more general learning research. There are several reasons for this. The first is that we live in an overwhelmingly one-size-fits all learning and working world. Secondly, most psychological research on cognition study group differences and hallmark experiments for measuring cognitive abilities are specifically designed to increase reliability of group level results. This makes them ill-suited for individual differences study (Rouder and Haaf, 2019). Similarly, most neuroimaging or EEG studies try to find significant differences in their experimental conditions by averaging across multiple subjects or sampling large amounts of data in few subjects.

But individual differences are *real*. Learners, given the same programming learning environment and the same amount of time will acquire different levels of skill at different rates (Prat et al., 2020). The best predictors for these individual differences, according to Prat et al. (2020) are other complex cognitive features like second language learning aptitude, numeracy, and fluid reasoning. It is not very clear where these individual differences arise if we consider skill learning from the point of view of more fundamental learning and memory mechanisms. Individual differences in working memory capacity are one source of these differences in learning outcomes but as described above, it is not clear where those capacity limitations occur.

All learners are a product of their genes and learning experience that influence their neural circuitry. And these determine meta-cognitive learning strategies, but it is difficult to test how these biases play out during learning. For example, progression in skill from declarative to procedural as described above is well studied both in humans and animal models. While complex, these systems normally work in parallel even if they compete at different stages of skill development (Poldrack et al., 2002; McDonald and Hong, 2013). It could be suggested, therefore, that for some individuals this typical progression of skill development might be interrupted through top-down executive influence or may be disproportionally reliant on strategies that favor sub-optimal learning mechanisms.